Introductory Topics in Statistical Learning

Eashan Gandotra

December 3, 2025

This document presents an overview of key topics in statistical learning, with an emphasis on the motivation and intuition behind models/concepts that is key to truly understanding them.

1 Models

1.1 Linear Regression

Linear regression finds the parameter β that minimizes the sum of squared residuals between the observed values y and the predicted values βx.

Problem: Given vectors $x, y \in \mathbb{R}^n$, find $\beta \in \mathbb{R}$ that minimizes

$$L(\beta) = \|y - \beta x\|^2 = \sum_{i=1}^{n} (y_i - \beta x_i)^2$$

Solution: To minimize $L(\beta)$, take the derivative with respect to β and set it to zero:

$$\begin{aligned} \frac{dL}{d\beta} &= \sum_{i=1}^{n} 2(y_i - \beta x_i)(-x_i) \\ &= -2\sum_{i=1}^{n} x_i y_i + 2\beta \sum_{i=1}^{n} x_i^2 \\ &= -2x^T y + 2\beta x^T x \end{aligned}$$

Setting the derivative to zero:

$$\begin{aligned} -2x^T y + 2\beta x^T x &= 0 \\ \beta x^T x &= x^T y \\ \beta^* &= \frac{x^T y}{x^T x} \end{aligned}$$

Note that $\beta^* = \frac{x^T y}{x^T x}$ can be written as $\frac{\text{Cov}(x,y)}{\text{Var}(x)}$ (up to scaling by n), which captures the idea that the slope should be proportional to how much x and y move together relative to how much x varies on its own.

This method has a closed-form solution with $O(n)$ time complexity, making it efficient and suitable for online learning, where the model can be updated incrementally as new data arrives. Although it assumes that y is a linear function of x, you can model nonlinear relationships by transforming the features first - polynomial regression, for example, uses features like $x, x^2, x^3, \ldots$ while still solving a linear regression on the transformed variables. This mix of efficiency, interpretability, and flexibility makes linear regression the go-to in fields like finance.

1.1.1 Ridge Regression

Ridge regression adds an L2 penalty term that shrinks coefficients toward zero but doesn't eliminate them:

$$L_{\text{ridge}}(\beta) = \|y - X\beta\|^2 + \lambda\|\beta\|^2$$

where $\lambda \geq 0$ controls the strength of regularization. It has closed-form solution:

$$\beta^* = (X^T X + \lambda I)^{-1} X^T y$$

1.1.2 Lasso Regression

Lasso adds an L1 penalty term that encourages sparsity by driving some coefficients to exactly zero:

$$L_{\text{lasso}}(\beta) = \|y - X\beta\|^2 + \lambda\|\beta\|_1$$

Lasso does not have a closed-form solution and requires iterative optimization methods to solve.

1.1.3 Elastic Net

Elastic Net combines both L1 and L2 penalties to get the benefits of both Ridge and Lasso:

$$L_{\text{elastic}}(\beta) = \|y - X\beta\|^2 + \lambda\left[\alpha\|\beta\|_1 + (1-\alpha)\|\beta\|^2\right]$$

where $\alpha \in [0, 1]$ controls the balance between L1 and L2 penalties.

1.2 Intuition for Regularization in Linear Regression

Both Ridge and Lasso improve the stability of coefficient estimates but introduce bias (see bias-variance tradeoff). Lasso is generally used to eliminate coefficients, while Ridge is better at dealing with collinearity among predictors. To understand why, we consider the constraint forms of the loss functions using Lagrange multipliers.

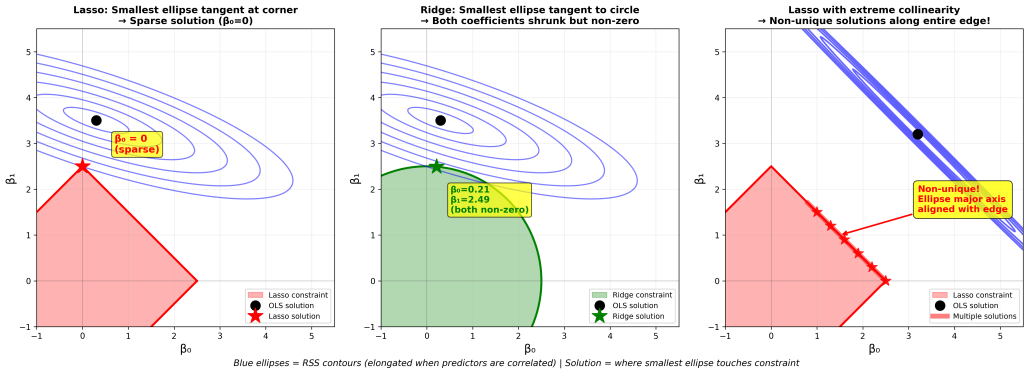
Instead of thinking of Lasso and Ridge as penalty functions, we can equivalently say we are minimizing the residual sum of squares (RSS) subject to constraints: $\|\beta\|^2 \leq t$ for Ridge or $\|\beta\|_1 \leq t$ for Lasso, for some $t > 0$. This creates useful geometric intuition.

Consider a simple case with only β_0 and β_1 on a plane where the coefficients are the axes. For a fixed RSS value, the points that satisfy $(y - X\beta)^2 = c$ for the constant c form an ellipse centered on the ordinary least squares solution. The Ridge constraint $\beta_0^2 + \beta_1^2 \leq t$

forms a circle, while the Lasso constraint $|\beta_0| + |\beta_1| \leq t$ forms a diamond with corners on the axes. The elastic net constraint is something in between the circle and diamond - a diamond with rounded corners.

To minimize RSS, we want the smallest ellipse that touches the constraint region. As we grow the ellipse outward from the origin until it touches the constraint, it's much more likely to hit a corner of the diamond than a specific point on the circle. Since the diamond's corners lie on the axes, they represent solutions where one coefficient is exactly zero. This geometric property explains why Lasso eliminates coefficients while Ridge shrinks them but rarely sets them to zero.

This also shows why Ridge is better at handling collinearity. When predictors are highly correlated, the RSS ellipses become elongated. For Lasso, the constraint diamond can intersect these elongated ellipses at multiple points along flat edges, leading to instability in which variables are selected. Ridge's circular constraint doesn't have this issue—it smoothly shrinks all correlated coefficients together, providing more stable solutions when features are collinear.



1.3 Logistic Regression

Logistic regression is used for binary classification, where we want to predict whether an observation belongs to class 0 or class 1 based on its features.

Problem: Given feature vectors $x_i \in \mathbb{R}^p$ and binary labels $y_i \in \{0, 1\}$ for $i = 1, \ldots, n$, find parameters $\beta \in \mathbb{R}^p$ that best predict the probability of each class.

Why not linear regression? We can't use standard linear regression for classification because it produces unbounded predictions - nothing prevents it from predicting values outside $[0, 1]$, which don't make sense as probabilities.

Solution: Instead of modeling probability directly, we model the log-odds (also called logit):

$$\log\left(\frac{P(y=1|x)}{P(y=0|x)}\right) = \beta^T x$$

This is a natural choice because log-odds are unbounded (unlike probabilities), making

them suitable for linear modeling. Additionally, each coefficient β_j has a clear interpretation as the change in log-odds per unit change in x_j, which keeps the model interpretable. Most importantly, assuming linearity in log-odds space is often more realistic than assuming linearity in probability space—it captures the intuition that the same change in a feature should have diminishing effects on probability as you approach the extremes (0 or 1), while having larger effects near the middle (0.5). This matches how many real-world binary outcomes behave, for example once you're already very likely to pass an exam, studying one more hour helps less than when you're on the borderline.

Deriving the sigmoid: Starting from the linear log-odds assumption, we can solve for the probability. Since $P(y = 0|x) = 1 - P(y = 1|x)$:

$$\log\left(\frac{P(y=1|x)}{1-P(y=1|x)}\right) = \beta^T x$$

Exponentiating both sides:

$$\frac{P(y=1|x)}{1-P(y=1|x)} = e^{\beta^T x}$$

Solving for $P(y = 1|x)$ we get:

$$P(y=1|x) = \frac{1}{1+e^{-\beta^T x}} = \sigma(\beta^T x)$$

This sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ automatically maps any real number to (0,1), is smooth and differentiable, and asymptotically approaches 0 and 1.

Interpretation: Each coefficient β_j tells us how much the log-odds changes per unit increase in feature x_j. In terms of odds, if we increase feature x_j by one unit, the odds $\frac{P(y=1|x)}{P(y=0|x)}$ are multiplied by e^{β_j}. For example, if $\beta_j = 0.5$, a one unit increase in x_j multiplies the odds by $e^{0.5} \approx 1.65$. If the odds were initially 2:1 (meaning $P(y = 1|x) = 2/3$), they become $2 \times 1.65 = 3.3 : 1$ (meaning $P(y = 1|x) \approx 0.77$). While the effect on log-odds is linear and the effect on odds is multiplicative, the effect on probability itself is nonlinear and depends on the starting probability.

Finding β via maximum likelihood: We can't directly observe probabilities - we only see outcomes (0 or 1). So we use maximum likelihood estimation: find the β values that make our observed data most likely. The likelihood of observing our data is:

$$L(\beta) = \prod_{i=1}^{n} P(y_i|x_i) = \prod_{i=1}^{n} \sigma(\beta^T x_i)^{y_i}(1-\sigma(\beta^T x_i))^{1-y_i}$$

Taking the log gives the log-likelihood:

$$\ell(\beta) = \sum_{i=1}^{n} \left[y_i \log(\sigma(\beta^T x_i)) + (1-y_i)\log(1-\sigma(\beta^T x_i))\right]$$

We maximize $\ell(\beta)$, or equivalently minimize the negative log-likelihood (cross-entropy loss):

$$L(\beta) = -\sum_{i=1}^{n} \left[y_i \log(\sigma(\beta^T x_i)) + (1 - y_i) \log(1 - \sigma(\beta^T x_i)) \right]$$

This loss function is prominent in information theory. Cross-entropy measures how well a predicted probability distribution matches the true distribution. Minimizing cross-entropy is equivalent to minimizing the KL divergence between predicted and true class probabilities.

There's no closed-form solution because the sigmoid makes this a nonlinear optimization problem, so we use iterative methods like gradient descent. Once trained, we classify by thresholding: predict $y = 1$ if $P(y = 1|x) > 0.5$, otherwise predict $y = 0$.

1.4 Bayesian Linear Regression

Standard linear regression finds an estimate β^* for our parameters. But this doesn't tell us anything about how confident we should be in our estimate. For example, if you have just 10 data points, you should be far less confident in your estimate than if you have 1000. That's where Bayesian linear regression comes in. It treats parameters as random variables with probability distributions, allowing us to quantify our uncertainty about the true parameter values. This is valuable when we have limited data, prior knowledge on what a parameter should be, and want to avoid overfitting.

The Bayesian approach: Instead of finding a single best value for β, we describe our beliefs about β using probability distributions. We start with a *prior distribution* $P(\beta)$ that encodes what we believe before seeing any data. Then, after observing data, we update our beliefs to get a *posterior distribution* $P(\beta|y, X)$ using Bayes' rule:

$$P(\beta|y, X) = \frac{P(y|X, \beta)P(\beta)}{P(y|X)}$$

So our beliefs on the distribution of β after seeing data (posterior) is proportional to how well the data fits (likelihood) times our beliefs before seeing data (prior). The mean of the posterior is the β used for prediction.

Setup: We assume the same linear model as standard regression: $y_i = \beta^T x_i + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, \sigma_\epsilon^2)$. For the prior, we'll use the common case where $\mu = 0$:

$$\beta_i \sim \mathcal{N}(0, \sigma^2) \text{ for all } i$$

This says we believe coefficients are likely small. The larger σ^2, the weaker this belief. As $\sigma^2 \to \infty$, we recover ordinary least squares. More generally, we could use any mean μ and variance, or even a different distribution. Gaussian distributions are a common choice because when both the prior and likelihood are Gaussian, the posterior is also Gaussian, giving us closed-form solutions.

Given parameters β, the likelihood of observing our data is:

$$P(y|X,\beta) = \prod_{i=1}^{n} \mathcal{N}(y_i|\beta^T x_i, \sigma_\epsilon^2) \propto \exp\left(-\frac{1}{2\sigma_\epsilon^2}\|y - X\beta\|^2\right)$$

Multiplying the Gaussian prior and likelihood gives a Gaussian posterior:

$$\beta|y, X \sim \mathcal{N}(\mu_{\text{post}}, \Sigma_{\text{post}})$$

where

$$\Sigma_{\text{post}} = \left(\frac{1}{\sigma_\epsilon^2}X^T X + \frac{1}{\sigma^2}I\right)^{-1}, \quad \mu_{\text{post}} = \frac{1}{\sigma_\epsilon^2}\Sigma_{\text{post}} X^T y$$

The posterior mean μ_{post} is our best point estimate. The covariance Σ_{post} captures our uncertainty - it shrinks as we get more data (larger $X^T X$), capturing the intuition that more data makes us more confident.

Connection to Ridge Regression: The posterior mean is exactly the ridge regression solution with $\lambda = \sigma_\epsilon^2/\sigma^2$. So ridge regression is actually Bayesian linear regression with a Gaussian prior. The difference is that Bayesian regression gives us the full posterior distribution Σ_{post}, not just a point estimate.

Making predictions: The predictive distribution at some point x_* is:

$$P(y_*|x_*, y, X) = \mathcal{N}\left(y_* \,\middle|\, \mu_{\text{post}}^T x_*,\ x_*^T \Sigma_{\text{post}} x_* + \sigma_\epsilon^2\right)$$

The mean prediction is $\mu_{\text{post}}^T x_*$, just like in standard regression. But the variance has two sources of uncertainty:

- $x_*^T \Sigma_{\text{post}} x_*$: We don't know the true β, so our predictions vary depending on which β values are plausible given the data we've seen
- σ_ϵ^2: Even if we knew β perfectly, there's still random noise in the observations

The key insight is that the first term changes depending on where x_* is. If we're predicting far from our training data, we haven't learned much about what β should be in that region, so this uncertainty is large. If we're predicting near training data, we're more confident about β, so this uncertainty shrinks.

1.5 Decision Trees

Linear models like regression make the strong assumption that y is a linear function of x. In reality the relationship is often more complex. For example, when recommending movies, you might want your algorithm to learn that if genre = comedy AND age < 25 predict high_rating. These kinds of hierarchical relationships are hard to capture with linear models. That's where decision trees come in.

Overview: A decision tree recursively partitions the feature space into regions, and makes predictions based on which region a data point falls into. At each node, we ask a yes/no

question about a feature (e.g. is age < 25?). Based on the answer, we follow the corresponding branch. Eventually we reach a leaf node that gives us a prediction.

Building a tree: We grow the tree greedily:

1. Start with all data at the root
2. Find the feature and threshold that best splits the data
3. Split the data into two child nodes based on this rule
4. Repeat recursively for each child node until some stopping criteria is met

What makes a split good? We want splits that make the data in each child node more similar. For regression, we typically minimize the sum of squared residuals within each region. For classification, we can use heuristics like:

- *Gini impurity*: $G = \sum_{k=1}^{K} p_k(1 - p_k)$ where p_k is the proportion of class k in the node. This measures the probability of incorrectly labeling if we randomly labeled an example according to the class distribution. It's zero when all examples are the same class, and maximum (0.5 for binary classification) when classes are evenly mixed.
- *Entropy*: $H = -\sum_{k=1}^{K} p_k \log(p_k)$, which also measures disorder/uncertainty in the node. Think of it as how many yes/no questions you'd need to identify an element's class. Pure nodes have zero entropy (no questions needed), while maximum mixing requires the most questions.

To choose the best split, we calculate the *information gain*: the reduction in impurity from parent to children. Specifically, we compute the weighted average impurity of the child nodes (weighted by the number of observations in each), and subtract this from the parent's impurity. The split that maximizes this information gain is selected.

Making predictions: For a new point x_*, we traverse the tree from root to leaf following the decision rules. For classification, we predict the majority class in the leaf node. For regression, we predict the average y value of all training points in that leaf.

Decision trees are interpretable and can handle non-linear relationships well, but are also prone to overfitting and high variance (small changes in data can create very different trees). Also worth noting that the greedy algorithm used to construct decision trees doesn't guarantee an optimal solution. To address the overfitting and reduce variance we can set a maximum depth for the tree, require a minimum number of samples to split/at each leaf, and prune the tree after building it.

1.6 Random Forest

Recall that while a decision tree is interpretable, it can be high variance and overfit; small changes in the data can result in very different trees. Random forests address this by building many trees and averaging their predictions. While individual trees may overfit in different ways, their errors tend to cancel out when averaged together.

The algorithm: Random forest builds B decision trees using two sources of randomness:

1. *Bootstrap sampling*: For each tree, randomly sample n training examples with replacement (some examples appear multiple times, others not at all). This creates B different training sets.

2. *Random feature subsets*: At each split in each tree, only consider a random subset of m features (typically $m = \sqrt{p}$ for classification or $m = p/3$ for regression, where p is the total number of features). This ensures trees aren't too correlated with each other.

After training all B trees, we make predictions by taking a majority vote across all trees for classification or averaging the predictions from all trees for regression.

Why does this work? Imagine you have B identically distributed random variables $X_1, \ldots, X_B$, each with variance σ^2 and pairwise correlation ρ. The variance of their average is:

$$\text{Var}\left(\frac{1}{B}\sum_{i=1}^{B} X_i\right) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

As $B \to \infty$, the second term vanishes, but the first term remains. This shows that:

- Averaging reduces variance (the $\frac{1}{B}$ term)
- But correlation between trees limits the benefit (the ρ term persists)

This is why we need both bootstrap sampling and random feature subsets—they decorrelate the trees (reduce ρ), allowing the averaging to be more effective.

Out-of-bag error: Since each tree is trained on a bootstrap sample, approximately 37% of the training data is left out for each tree. These out-of-bag (OOB) data can be used as a validation set - we can predict using only trees that didn't see it during training. The OOB error provides an honest estimate of test error without needing a separate validation set.

Feature importance: Random forests can measure feature importance by tracking how much each feature decreases impurity when used for splitting, averaged across all trees. Alternatively, we can measure the sensitivity of the OOB error to permuting a feature's values. Important features cause large increases in error when permuted.

Comparison to single trees: Random forests sacrifice the interpretability of single decision trees (it's harder to understand why a prediction was made when it comes from hundreds of trees), but are lower variance and more generalizable. They're also more robust to outliers and noisy features since errors from individual trees average out.

Random forests are widely used in practice because they often work well with minimal tuning, making them a good baseline for problems where you don't need much interpretability.

1.7 Gradient Boosted Trees

Gradient boosting builds trees sequentially, where each new tree learns to correct the mistakes of all previous trees combined. This method is another workhorse of industry and is commonly used in fields like quantitative finance.

The algorithm: We start with a simple initial prediction, like the mean of target values. Then we iteratively:

1. Calculate residuals: $r_i = y_i - F_{m-1}(x_i)$ where F_{m-1} is the prediction made by the previously trained set of trees (the current *ensemble*)
2. Train a new tree h_m to predict these residuals
3. Add the tree to our ensemble: $F_m(x) = F_{m-1}(x) + \alpha \cdot h_m(x)$

The learning rate α (typically 0.01-0.3) controls how much each tree contributes. Smaller values require more trees, but often generalize better.

Where does the gradient come in?: Gradient boosting is actually performing gradient descent in function space. For a loss function $L(y, F(x))$, we want to find the function F that minimizes the expected loss. At each step, we compute the negative gradient of the loss with respect to our current predictions:

$$r_i = -\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}$$

For squared loss, this gradient is simply the residual $y_i - F_{m-1}(x_i)$. We fit new trees to approximate the negative gradient.

In practice: Libraries like XGBoost, LightGBM, and CatBoost implement optimized versions of gradient boosting with additional features like regularization, histogram-based splitting for speed, and handling of categorical variables.

1.8 k-means Clustering

Unlike **supervised learning** where we have labeled data to train on, clustering is an **unsupervised learning** problem where we want to discover structure in unlabeled data. The goal is to group similar observations together and separate dissimilar ones. k-means is one of the most popular clustering algorithms.

Problem: Given data points $x_1, \ldots, x_n \in \mathbb{R}^p$ and a choice of k, partition the data into k clusters such that points within each cluster are similar to each other.

The algorithm: k-means minimizes the within-cluster sum of squares (WCSS):

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

where C_i is the i-th cluster and μ_i is its centroid (mean). The algorithm alternates between two steps:

1. *Assignment step*: Assign each point to the nearest centroid: $C_i = \{x : \|x - \mu_i\| \leq \|x - \mu_j\| \text{ for all } j\}$
2. *Update step*: Recalculate the centroids as the mean of the assigned points

We repeat until the centroids stop moving. Each step is guaranteed to decrease or maintain the WCSS, so the algorithm converges to a local minimum.

Choosing k: The number of clusters k must be specified in advance. Common approaches include:

- *Elbow method*: Plot WCSS vs k. The elbow (where the rate of decrease slows) suggests a good k
- *Silhouette analysis*: For each point, compare its distance to points in its own cluster vs. other clusters. Higher silhouette scores indicate better clustering

Initialization: The result of k-means is heavily dependent on the initial centroid placement. Common initialization strategies include random initialization (run k-means multiple times and pick best result) and k-means++ (choose initial centroids that are far from each other).

Limitations: k-means assumes clusters are spherical and roughly equal in size, since it minimizes distance to centroids. It struggles with elongated, irregular, or overlapping clusters. The algorithm is also sensitive to outliers, which can pull centroids away from the true cluster centers.

1.9 Support Vector Machines

Unlike logistic regression, which maximizes likelihood, Support Vector Machines (SVMs) take a geometric approach - they find the decision boundary that maximizes the margin between classes.

Intuition: Imagine two classes of points on a plane. Infinitely many lines could separate them, but SVMs choose the line that stays as far as possible from both classes. This maximum margin principle leads to better generalization. If future data points differ slightly from training data, a large-margin boundary is less likely to misclassify them.

Problem setup: Given training data (x_i, y_i) where $x_i \in \mathbb{R}^p$ and $y_i \in \{-1, +1\}$, we want a hyperplane $w^T x + b = 0$ that separates the classes with maximum margin. The distance from point x_i to the hyperplane is $\frac{|w^T x_i + b|}{\|w\|}$, and correctly classified points satisfy $y_i(w^T x_i + b) > 0$.

The optimization problem: To maximize the margin, we maximize the distance to the closest points (the support vectors):

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w^T x_i + b) \geq 1 \text{ for all } i$$

The constraint ensures all points are correctly classified with margin at least $\frac{1}{\|w\|}$. Minimizing $\|w\|^2$ maximizes this margin. Only support vectors - points exactly on the margin boundaries - affect the solution. Other points could move freely without changing the optimal hyperplane.

Soft margin SVM: Real data is rarely perfectly separable. We introduce slack variables $\xi_i \geq 0$ allowing some margin violations:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i \quad \text{subject to} \quad y_i(w^T x_i + b) \geq 1 - \xi_i$$

The parameter C controls the tradeoff: large C penalizes violations heavily (risking overfitting), while small C allows more violations for a larger margin.

The kernel trick: What if no hyperplane can separate the classes? The key insight is that data which isn't linearly separable in the original space might become separable in a higher-dimensional space. Consider the classic XOR problem: points at $(0, 0)$ and $(1, 1)$ belong to one class, while $(0, 1)$ and $(1, 0)$ belong to another. No line separates them. But if we add a new feature $x_1 x_2$, the points become $(0, 0, 0)$, $(1, 1, 1)$, $(0, 1, 0)$, $(1, 0, 0)$ - now a plane can separate them.

Explicitly computing high-dimensional mappings $\phi(x)$ can be expensive or even impossible (some mappings are infinite-dimensional). The kernel trick avoids this: since SVMs only need inner products between points, we use kernel functions $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ that compute these inner products directly without ever constructing ϕ. Common kernels include:

- *Linear*: $K(x_i, x_j) = x_i^T x_j$ (no transformation)
- *Polynomial*: $K(x_i, x_j) = (x_i^T x_j + 1)^d$ (captures feature interactions up to degree d)
- *RBF (Gaussian)*: $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$ (infinite-dimensional mapping; acts as a similarity measure where nearby points have high kernel values)

SVMs work well in high dimensions and are memory efficient since only support vectors matter. However, they require careful feature scaling, don't naturally provide probability estimates, and can be slow to train on large datasets.

2 Evaluation

2.1 Cross-Validation

Cross-validation is used to see how well a model generalizes. There a few common types:

K-fold cross-validation: Divide data into k equal folds. For each fold, hold it out as validation, train on the remaining $k-1$ folds, and evaluate performance. Average the k results for a robust generalization estimate. Common choices are $k = 5$ or $k = 10$.

Stratified cross-validation: For classification problems with imbalanced classes, stratified CV ensures each fold maintains the same class distribution as the original dataset.

Time series considerations: Standard cross-validation randomly assigns data points to folds, which, for time series data, could let you train on future observations to predict past ones. Time series cross-validation needs to respect ordering. A common approach is rolling window validation, where you train on some rolling window of data and validate using a portion of the subsequent data.

2.2 Model Evaluation Metrics

Common evaluation metrics include:

Classification metrics

- *Accuracy*: Fraction of correct predictions. This can be misleading if classes are imbalanced
- *Precision*: Fraction of predicted positives that are true positives
- *Recall*: Fraction of true positives that were predicted correctly
- *F1-score*: Harmonic mean of precision and recall (useful when you need to balance both)

Regression metrics

- *Mean Squared Error (MSE)*: Average of squared differences between predicted and true values
- *Mean Absolute Error (MAE)*: Average of absolute differences. This is more robust to outliers than MSE
- *R-squared*: Fraction of variance explained by the model

3 Other Concepts

3.1 Bias-Variance Tradeoff

When training a model, we care about performance on new data. For squared loss, the prediction error on new data has three components: bias, variance, and irreducible error.

Bias measures how far our model's average prediction is from the true value. High bias means the model is too simple—it makes assumptions that don't match reality. For example, fitting a line to quadratic data has high bias.

Variance measures how much predictions change across different training sets. High variance means the model is too flexible—it fits noise rather than the underlying pattern. A deep decision tree might perfectly fit each training set but give wildly different predictions.

The tradeoff: As model complexity increases, bias decreases (can fit more patterns) but variance increases (more sensitive to training data).

Mathematically:

$$\text{Expected Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

3.2 Handling Missing Data

Deletion: The simplest approach is to remove any features with missing data.

Imputation: Fill in missing values with reasonable substitutes. For numerical data, we can use the mean (if roughly normal) or median (if skewed or has outliers). For categorical data, we can use the mode (most frequent category) or create a new unknown category.

3.3 Hyperparameter Optimization

While model parameters (like regression coefficients) are learned from data, hyperparameters are choices we make beforehand that control how learning happens.

Grid search: Define ranges for each hyperparameter and try every combination. This approach is thorough but computationally expensive.

Random search: Instead of trying all combinations, sample randomly from each hyperparameter's range. This often finds nearly optimal solutions much faster than grid search.

Bayesian optimization: Instead of randomly trying hyperparameters, methods like Optuna learn from previous trials to make smarter choices by creating an internal model to predict how well different hyperparameters will perform.

3.4 Dimensionality Reduction

Modern datasets often have many features, which can make training models computationally expensive, visualization difficult, and lead to the curse of dimensionality. Dimensionality reduction addresses these issues by finding lower dimensional representations of the features that preserve the most important information.

The curse of dimensionality: As dimensions increase, data points become increasingly sparse - imagine how much more space there is to spread out in a 1000-dimensional cube versus a square. This sparsity causes distances between points to become nearly uniform (everything is far from everything else), making nearest neighbors less meaningful. Many

machine learning algorithms that rely on local similarity break down because in high dimensions, your nearest neighbor might still be quite different.

Principal Component Analysis (PCA): PCA is the most widely used dimensionality reduction technique. Imagine data scattered in a cloud. PCA finds the axes along which this cloud is most stretched out. The first principal component points in the direction of maximum stretch (variance), the second component (orthogonal to the first) captures the next most stretch, and so on. Think of it as fitting the tightest possible box around your data points - the edges of this box are your principal components.

Mathematically, PCA finds the eigenvectors of the covariance matrix $\Sigma = \frac{1}{n}X^T X$ (where X is centered data). These eigenvectors are the directions where data varies most, and their corresponding eigenvalues tell you how much variance each component captures. If the first few eigenvalues are much larger than the rest, you can keep only those components and still capture most of the information.

Algorithm:

- Center the data by subtracting column means, so the origin represents the average point
- Compute the covariance matrix $\Sigma = \frac{1}{n}X^T X$, which captures how features vary together
- Find eigenvalues and eigenvectors of Σ
- Sort eigenvectors by descending eigenvalue magnitude. Eigenvalues quantify variance along each direction
- Project data onto the top k eigenvectors to get a k-dimensional representation preserving the most variance

PCA's linear transformation makes it fast and stable, while the orthogonal components help eliminate multicollinearity. The explained variance ratio tells you exactly how much information you retain, and components remain interpretable as linear combinations of original features. However, PCA only captures linear relationships and maximizes variance, rather than predictive power, for your specific task. Outliers can heavily influence results, and since all original features contribute to each component, you don't achieve true feature selection.

4 References

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.). Springer.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning: With Applications in R*. Springer.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.